[redacted]

Math

Report No. UIUCDCS-R-74-678

COST MINIMIZATION IN COMPUTER SYSTEMS SUBJECT TO MULTIPLE MEMORY CONSTRAINTS

by

Michael Austin Jamerson

October 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-74-678

COST MINIMIZATION IN COMPUTER SYSTEMS SUBJECT TO MULTIPLE MEMORY CONSTRAINTS

by

Michael Austin Jamerson

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1974.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Western Electric and their Engineering and Science Fellowship Program whose financial assistance helped make this paper possible. In addition, I would like to thank Dr. Edward K. Bowdon whose guidance and advice in preparation of this paper was invaluable. Lastly, deepest thanks to my loving wife who provided constant support and encouragement throughout the preparation of this paper.

TABLE OF CONTENTS

1. INTRODUCTION

Every computer center is faced with the problem of providing service to its customers at a cost and speed which is acceptable to the customer and at the same time satisfies management goals. One important aspect of the cost of a computing center is the amount and types of memory which are available. Generally, the types of memory devices are dictated by the user community. The amounts of the memories, on the other hand, are limited if not by cost, then by physical limitations such as floor space or hardware design. Whenever a resource is limited to a degree that requires considerations of the size of the resource, then that resource is constrained. It is the goal of this paper to present scheduling techniques which minimize the amounts of idle resources, while providing nearly optimal flowtimes.*

The systems which we will consider will be those with a single central processing unit (CPU) having the capability of concurrently processing two or more jobs. We will use the term multiprogramming to apply to these systems. The essential point in a multiprogramming system is that although more than one job may be present in main memory (CORE) only a single job is being processed at any given instant. The flowtime or run time of a job is derived from three elements: 1) active time, 2) passive time, and 3) wait time. Active time is the time in which CPU processing takes place. The term passive time will be used to denote the time consumed by events which are external to the CPU such as input-output processing. Time which is neither active nor passive will be called wait time. Wait time includes but is not limited to time spent in queues prior to the start of processing and time spent waiting for the CPU when the CPU is processing another job.

*Flowtime is defined as the amount of time a job spends in the system.

Scheduling techniques which have been previously discussed in the literature are generally concerned with the problem of minimizing some measure such as maximum flowtime, average flowtime, or maximum lateness [1]. Results for systems where a single server processes a single request at a time also have been extensively examined. Browne, Lan, and Baskett [2] discuss the results of various scheduling strategies such as first-come-first-serve (FCFS) and shortest processing time (SPT) in a multiprogrammed environment. In all of the above strategies, jobs are selected for processing on the basis of their arrival or required processing time. One technique considered by Browne, Lan and Baskett [2], the smallest memory first (SMF) algorithm, is concerned with memory availabilities, but not with the best use of those resources.

Some scheduling systems currently in use are based on a scheme for evaluating the resources requested by a job. In these systems the resource requests such as amount of CORE and CPU processing time are used as the independent variables of a function. The result of evaluating this function is then used to assign a priority to the job. Other systems rely on a user defined priority and operate under a FCFS selection discipline within the individual priority queues. In all of the techniques which have been discussed the resources are checked to insure that enough of the resource is available to satisfy the requirements of a given job. However, no attempt is made to select jobs which best utilize the available resources.

Denning [3] presents one approach to the problem of efficient utilization of the available resources. He presents the concept of a "working set" and suggests that jobs be run according to their ability to restore system balance as opposed to using some other priority criteria. This approach merits further discussion and will be examined in Chapter 3.

A selection technique which attempts to maximize the utilization of the available resources by the selection of one or more jobs for execution will be called a best fit selection technique (BFST). A BFST may be used in conjunction with some other technique such as SPT or FCFS in order to minimize some other measure of the system, but the primary goal of a BFST will always be the minimization of the amount of idle resources.

It has been shown that when arrival times are exactly known, that inserted idle time may improve the resource utilization of a system [1]. The concept of inserted idle time may be incorporated into a BFST. We will not consider cases in which the arrival times are known and therefore we will not be concerned with inserted idle time.

In Chapter 2 we will define the various types of resources and develop a methodology for determining the effect of job interaction in a multiprogramming system on a given job's run time. These results are then used in Chapter 3 to develop a BFST for the case of a system with a single constrained memory resource. Chapter 4 will investigate the application of a BFST to a generalized system with multiple constrained memory resources. Finally we conclude with a review of the results of Chapters 3 and 4 and then attempt to identify the future research which will be required.

2. INTERACTION EFFECTS IN A MULTIPROGRAMMED SYSTEM

2.1 Memory Resources

Every computer center is faced with the problem of meeting its customers' demands for service. These service demands consist of two parts: (1) requests for portions of the available memory resources and (2) a period of time during which those resources will be used. The responsibility of meeting these demands in a time frame which is acceptable to the users rests with the computing center management. Management must, on the other hand, attempt to balance the use of these resources and minimize the amount of idle resource. We now turn our attention to the subject of what those resources are and how multiple users interact in the pursuit of the use of those resources.

We will consider each computer center to consist of a single CPU which is capable of multiprogramming and a number of constrained memory resources. In considering the multiprogramming capability of the CPU, we will assume that neither the hardware nor its controlling operating system will pose a limit on the number of jobs which may run concurrently.

The constrained memory resources may be divided into two classes: 1) contiguous and 2) fragmented. A contiguous memory resource (CMR) is a resource in which a contiguous portion of the resource must be used to satisfy a resource request. The prime example of a contiguous memory resource is the CORE of the CPU. Here the user's request must be filled by s single contiguous portion of CORE. There are cases in which CORE is not treated as a contiguous resource, but those cases will not be considered here. Fragmented memory resources (FMR) are those resources in which requests for the resources need not be filled by a contiguous assignement but rather by the assignment of individual resource units. An example of an FMR is the magnetic tape drive pool. In this case, a user's request for a number of tape drives is satisfied by selecting as many of the unassigned drives as are required. The question of whether the drives are logically or even physically contiguous is not raised.

The available size of any memory resource may be reduced from the actual size. These reductions may be static or dynamic in nature. We will call those reductions which occur because of various day-to-day requirements of the computing center, static reductions. Examples of static reductions include the decrease in available CORE caused by the presence of the operating system and other real time activities or the reduction in available tape drives resulting from the allocation of one or more drives for the function of logging system activities. Static reductions then are predictable and can be taken into consideration. Dynamic reductions, on the other hand, are unpredictable in terms of occurrence and size. Particular examples of dynamic reductions are equipment failures or the dedication of a portion of CORE to insure the completion of some critical job.

Thus we see that, at any given instant, the amounts of the memory resources may be less than originally planned. For convenience we will view the static reductions of each resource as though those portions of the resources were never available. Dynamic reductions will be treated as service requests of an indefinite length which exactly equal the size of the particular dynamic reduction. We will denote the available amount of a resource i, given by the difference of the original size of the resource and the sum of the static reductions for that resource, as R_i.

2.2 Job Interaction

Each user job which enters the computing center and begins processing interacts with other jobs which are in the center. This interaction results in a prolongation of the flowtime for all of the jobs which are involved. In this section we will develop analytical tools which will provide information concerning this prolongation.

The systems with which we will be concerned are single processor systems. A basic element in our concept of a single processor is that it may only process a single task at a time. Thus, even in a multiprogramming environment in which many jobs may be in execution and all are competing for the CPU, only one job is using the CPU at any given instant.

Each job which enters the computing center will be characterized by a set of parameters which indicate the resources required by that job and the processing characteristics of that job. These parameters are:

1) a_i - the amount of CPU processing time required by job i.
2) p_i - the amount of passive time which job i requires.
3) R_{ij} - the amount of resource j which is required by job i.

When job i is the only job which is in execution, then the execution time, e_i, for job i is the sum of the active and passive times

$$e_i = a_i + p_i \ . \qquad (1)$$

In dealing with the jobs which are characterized by the n+2 tuple $(a_i, p_i, R_{i1}, R_{i2}, \cdots, R_{in})$ we assume that:

1) the passive time, p_i, is evenly distributed over the execution time, or that in any time interval, t, the amount of active time is $\frac{ta_i}{e_i}$ and the amount of passive time is $\frac{tp_i}{e_i}$.

2) the amount, R_{ij}, of any resource which is requested by job i is less than the available amount, R_j, of resource j.

When two or more jobs are concurrently executing we observe a phenomenum which will be called overlap. Overlap is a result of the use of the CPU to process a job when some other job is in a period of passive time. The effect of overlap is a reduction in the overall execution time. From this we see that overlap is a measure of how well the active and passive periods of jobs mesh. Using these observations we will define the overlap factor, $\mathcal{O}$, as the ratio of the amount of processing that occurs in a given period of time. In dealing with overlap factors we will make the following assumptions:

1) the amount of overlap in a given period is constant, $0 < \mathcal{O} \leq 1$, when two or more jobs are processing and unity when only one job is processing.

2) processing is equitably rationed to each job which is being processed.

The amount of overlap is dependent on factors which are intrinsic to the computer center and is to be determined experimentally. In real life the amount of overlap is dependent on the total number of jobs. Beginning at some low value, $\mathcal{O}$ increases to some optimum, and then decreases as more jobs compete for the resources of the CPU.

As stated the execution time, e_i, of a job is dependent on both the job and the other jobs in the system. We now develop the techniques for determining the relationship of the execution time to the state of the system. Let $\mathcal{O}(n)$ be the overlap factor for a given system with n jobs. Let j_i be a job with active time, a_i, and passive time, p_i. The execution time for j_i is $e_i \geq a_i + p_i$, where the equality holds if j_i is the only job which is processed.

We assume that the available processor time is rationed in an equitable fashion to every job which is being concurrently processed. This rationing is designed to prevent the disproportionate assignment of processing time that could occur if a job with a large amount of active time and little or no passive time were allowed to execute until it relinquished control of the CPU. Thus we can guarantee that each of the n jobs which are processing will be provided with at least $\frac{1}{n}$ of each available processor minute. Under our assumption that the passive time is evenly distributed over the execution time, we see that no job may use more than $\frac{a_i}{e_i}$ of a processor minute.

If a job cannot utilize its entire share of the processor, $\frac{a_i}{e_i} < \frac{1}{n}$, the unused time will be distributed evenly to the other jobs which are executing. Thus, each job is given an initial allocation of $t_i \leq 1$ minutes of the available processor time, where $t_i = \min(\frac{1}{n}\Theta, \frac{a_i}{e_i})$. The unused time, U_T, is given by the expression

$$\begin{aligned} U_T &= \sum_{i=1}^{n} \frac{1}{n}\Theta - \min(\frac{1}{n}\Theta, \frac{a_i}{e_i}) \\ &= \Theta - \sum_{i=1}^{n} t_i \quad . \end{aligned} \tag{2}$$

The unused time, U_T, may now be allocated to those $n' \leq n$ jobs having $t_i < \frac{a_i}{e_i}$. The new allocations are then

$$t_i = \min(t_i + \frac{U_T}{n'}, \frac{a_i}{e_i}) \ . \tag{3}$$

Again the unused time is colllected and distributed. This process continues until each job has been allocated $t_i = \frac{a_i}{e_i}$ units of time or $U_T = 0$.

The above discussion indicates the technique used to determine the amount of processing which is allocated to each concurrently running job. Using this technique under the assumption that the passive time is evenly distributed with respect to the processing time we may now calculate the real time requirements for processing a set of jobs. The following example is provided to illustrate these techniques.

Let us start with four jobs which are to be processed by a system with an overlap factor $\sigma = 0.9$. To avoid confusion we will not be concerned with resource constraints on requirements at this time. The parameters of the four jobs are shown in Table 2.1. In this illustration, we assume that jobs 1, 2, and 3 are all started at the same time, T = 0, and that job 4 will begin execution when one of the first three jobs completes.

Table 2.1

Job	active time, a_i	passive time, p_i	$\frac{a_i}{a_i+p_i}$
1	1	0	1
2	2	2	.5
3	1	3	.25
4	1	2	.33

Using the information in Table 2.1 and the preceding formulas we will now determine the processor allocations. Table 2.2 shows the results of the calculations and illustrates the effect of multiprogramming. In the Gantt chart of Figure 2.1 we have graphically shown the progress of each job. From the above description we see that the CPU allocation factor, τ, may change whenever a job begins or finishes processing. It is these points in time which must be determined. Barring any changes in the system a given job will require $\frac{a_i}{\tau_i}$ to finish. Thus we may determine the completion time of the first job by computing $\frac{a_1}{\tau_1} = 3.077$ to be the smallest and so job 1 is the first to finish. Using this value we may now determine how much processing the other jobs received by calculating $\tau_i(\frac{a_1}{\tau_1})$. We see these values tabulated in column 7 of

Table 2.2. A sample Job Stream

Time	Job	$\frac{1}{n}\Theta$	$\frac{a_i}{e_i}$	CPU allocation (τ)	time remaining	time processed
0	1	.3	1.0	0.325	1.0	0
	2	.3	0.5	0.325	2.0	0
	3	.3	0.25	0.25	1.0	0
3.0769	1	.3	1.0	0.325	0	1.0
	2	.3	0.5	0.325	1.0	1.0
	3	.3	0.25	0.25	0.2308	0.7692
3.0769	2	.3	0.5	0.325	1.0	1.0
	3	.3	0.25	0.25	0.2308	0.7692
	4	.3	0.33	0.325	1.0	0
4.0000	2	.3	0.5	0.325	0.7000	1.3000
	3	.3	0.25	0.25	0	1.0
	4	.3	0.33	0.325	0.7000	0.3000
4.0000	2	.45	0.5	0.5	0.7000	1.3000
	4	.45	0.33	0.33	0.7000	0.300
5.4000	2	.45	0.5	0.5	0	2.0
	4	.45	0.33	0.33	0.2333	0.7667
5.4000	4	1.0	0.33	0.33	0.2333	0.7667
6.1	4	1.0	0.33	0.33	0	1.0

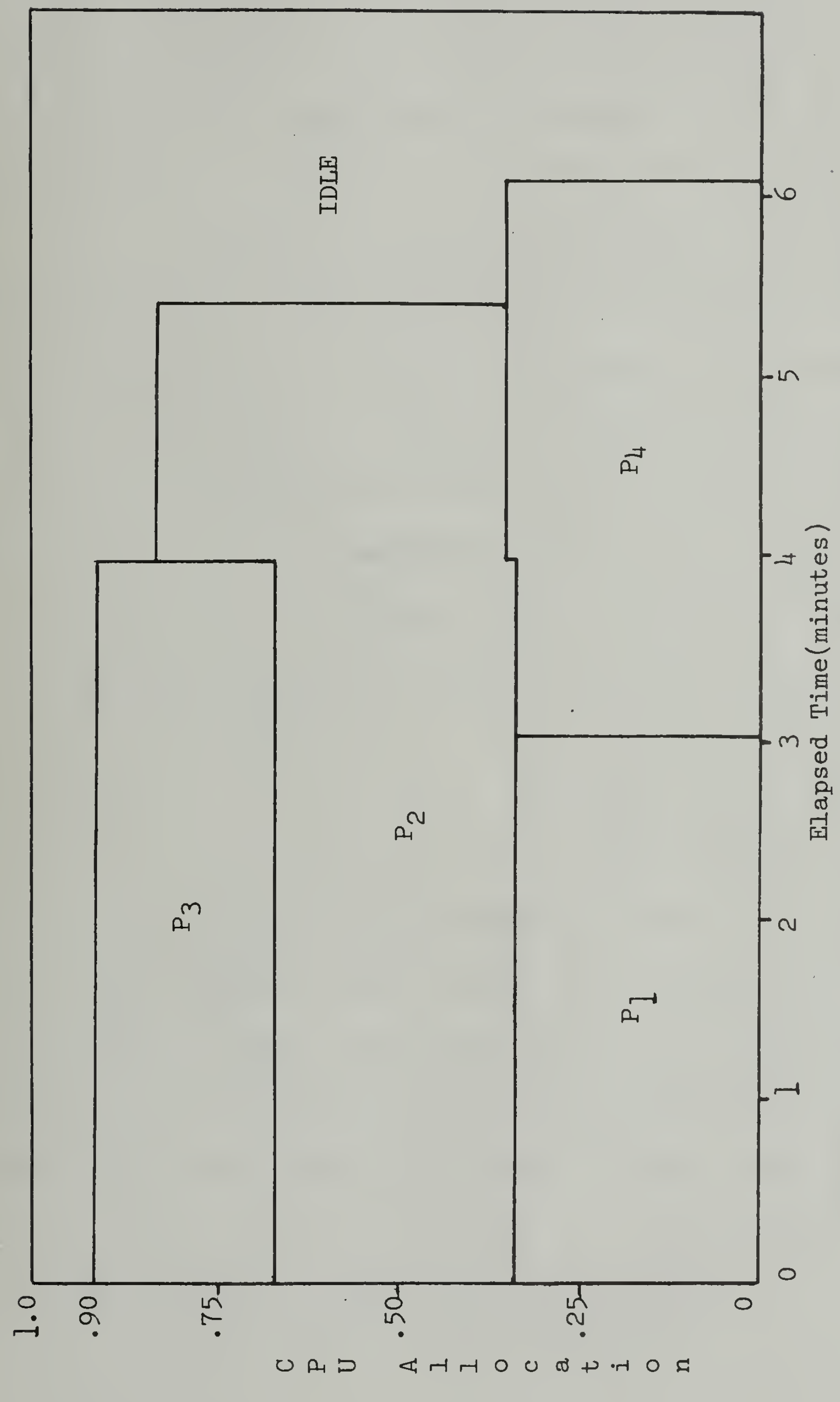

FIGURE 2.1 Gantt chart of one schedule for jobs in Table 2.1

Table 2.2. To locate the next completion point, we repeat the procedure of finding the smallest $\frac{a_i}{\tau_i}$, keeping in mind that each a_i will now have a new value equal to the original value minus the amount of processing accomplished. Thus we see that at each termination we must recalculate the CPU allocation factor, τ_i, for each job and subsequently determine the next termination point given by $\min_{\forall i}(\frac{a_i}{\tau_i})$. Having carried out this procedure we find the time taken to complete the four jobs is approximately 6.1 minutes.

A question that we might ask at this point is what is the minimum time to complete the four jobs. By summing the active time requirements shown in Table 2.1, we find that five minutes of processor time are needed. Since the CPU may only process one job at a time this is the minimum time needed for the four jobs. But due to the constraint that the overlap factor is 0.9 only 0.9 of every minute will be used to process the jobs. The minimum flowtime, F_{min}, now becomes the sum of the active times for each job divided by the overlap factor or

$$F_{min} = \sum_{i=1}^{n} a_i/\mathcal{O} . \qquad (4)$$

For this case, $F_{min} = 5.55$. This minimum only holds true when the sum of the CPU allocation factor is equal to the overlap factor at every instant during the run. This condition, $\Sigma\, \tau_i = \mathcal{O}$ can be satisfied. This is satisfied by processing jobs 2,3,4 together and then starting job 1 when job 4 finishes. This is shown in the Gantt chart of Figure 2.2. Another measure of interest is the average flowtime defined:

$$\overline{F} = \frac{1}{n} \sum_{i=1}^{n} a_i + p_i + \omega_i . \qquad (5)$$

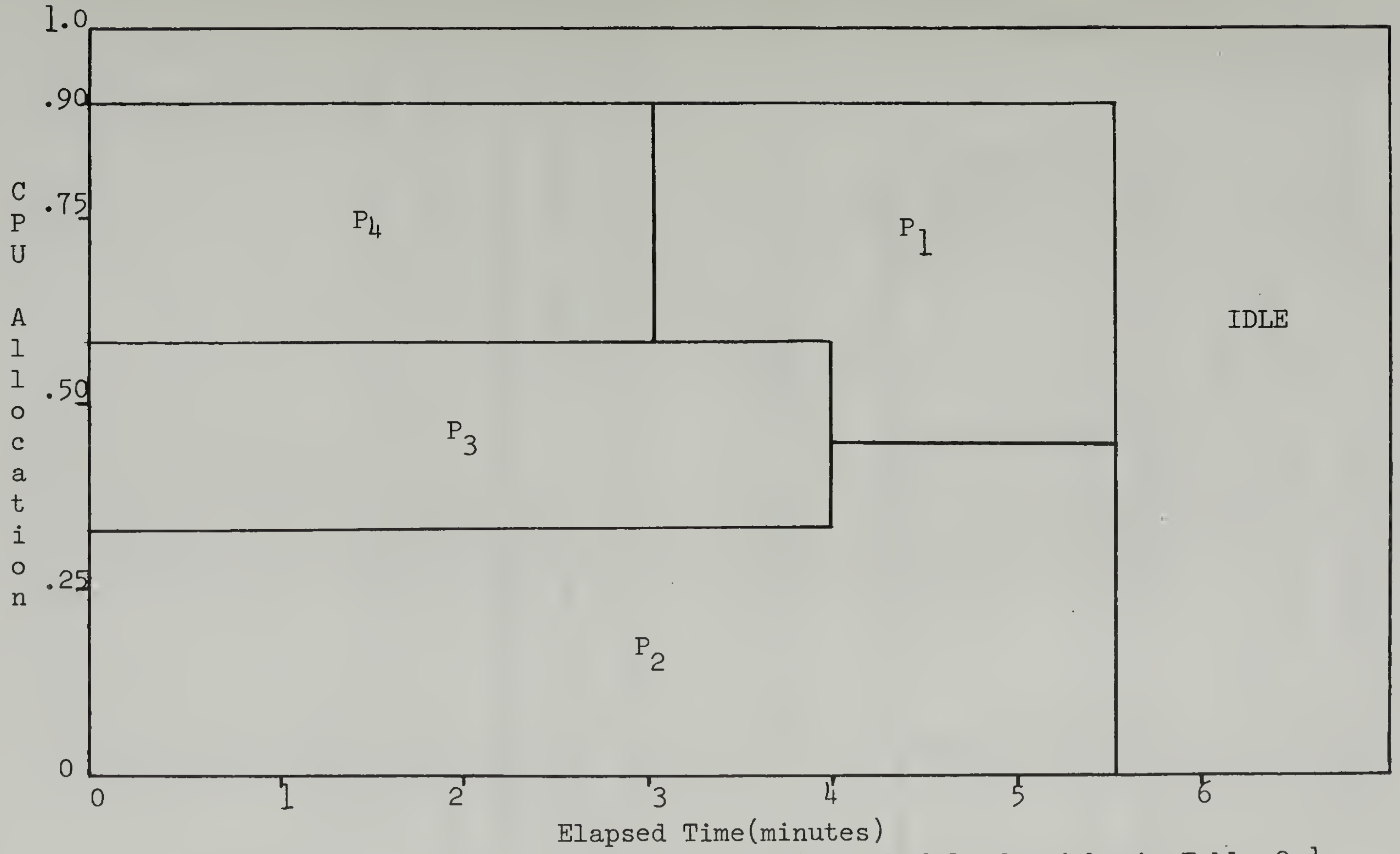

FIGURE 2.2 Gantt chart of minimumm flowtime schedule for jobs in Table 2.1

As shown in Conway, Markwell, and Miller [1] schedules which result in minimum flowtime, F_{min}, may not result in minimum average flowtime, $\bar{F}_{min}$. For the set of jobs shown in Table 2.1, $\bar{F}_{min}$ and F_{min} are both satisfied by the schedule given in Figure 2.2. Table 2.3 lists a set of jobs where the schedule which results in $\bar{F}_{min}$ does not result in F_{min}. These two schedules are presented in the Gantt charts of Figures 2.3 and 2.4.

Job	Active Time	Passive Time
1	1	1.5
2	1	2.33
3	2	0
4	2	1.33
5	2	4

Table 2.3. A set of jobs where schedule for $\bar{F}_{min}$ is not the same as F_{min} schedule.

We have seen in this chapter that the memory resources of a computing center may be divided into two classes, contiguous and fragmented. In the second section we examined the question of how job interaction affects the execution time of other concurrently running jobs. Armed with these tools we are now prepared to attack the problem of scheduling jobs in a system with constrained memory resources.

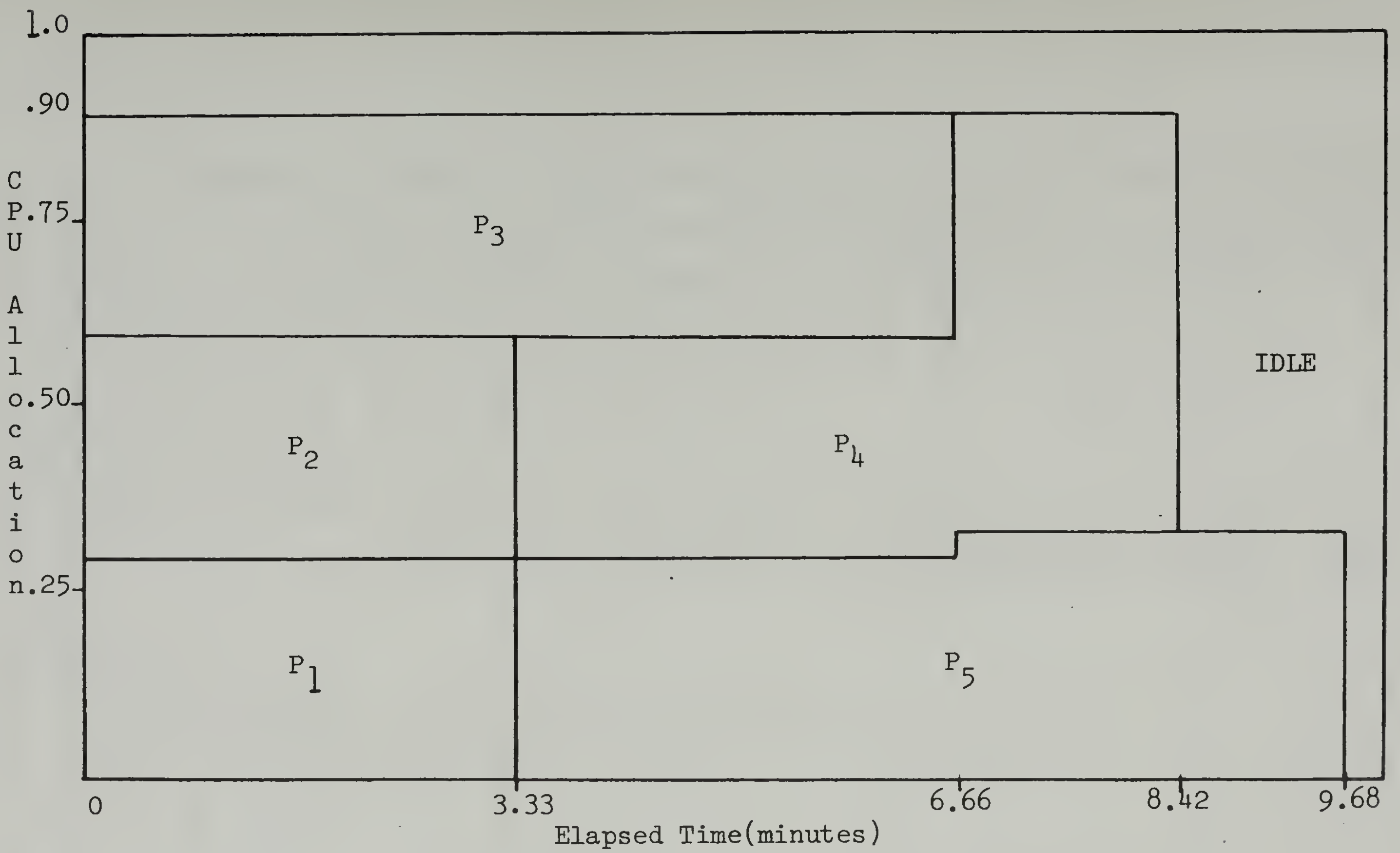

FIGURE 2.3 A minimum average flowtime schedule for jobs in Table 2.3

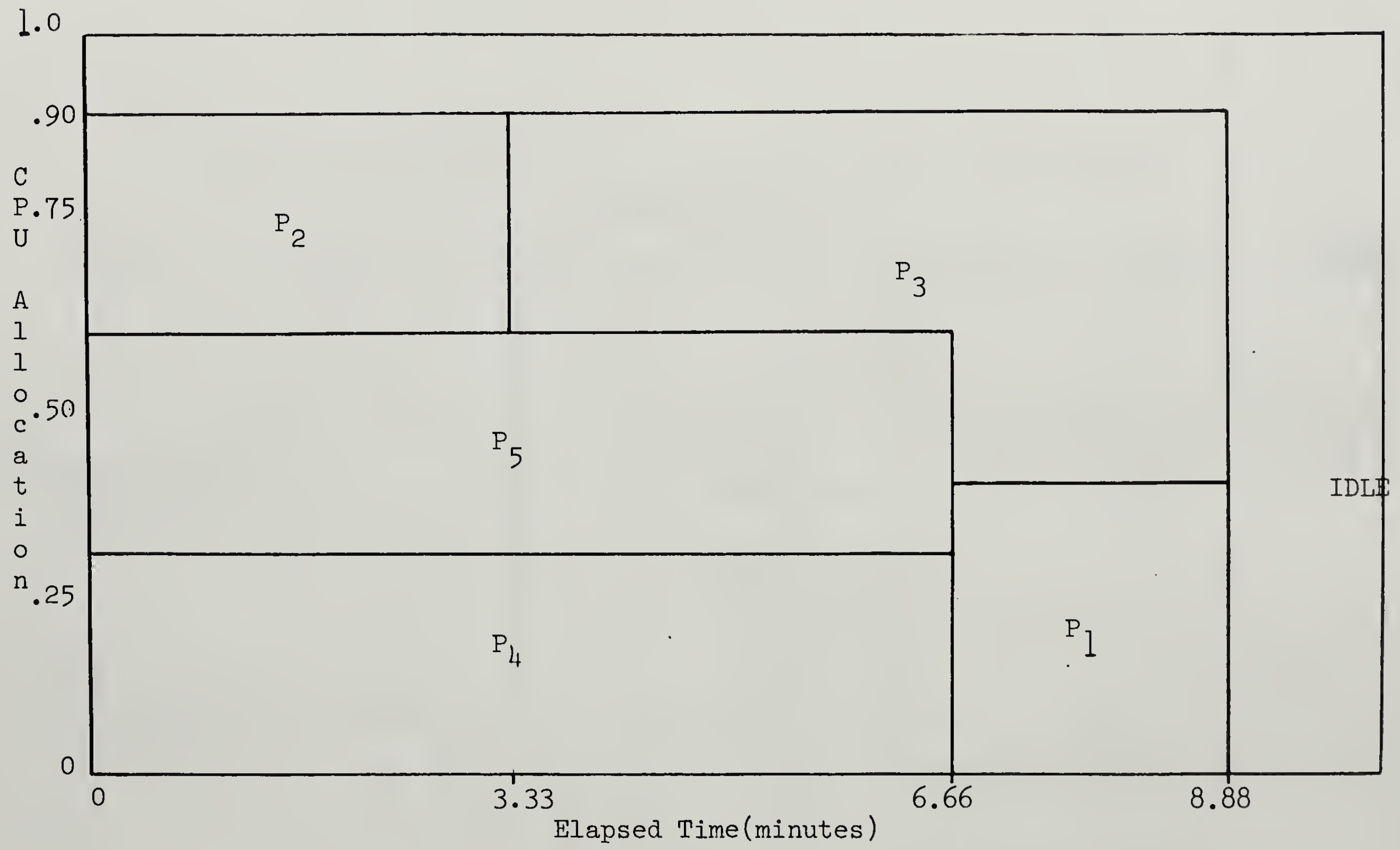

FIGURE 2.4 A minimum maximum flowtime schedule for jobs in Table 2.3

3. SINGLE RESOURCE SYSTEMS

We now turn our attention to the development of a best fit scheduling technique for a system with a single constrained resource. Methods and results will be developed for both contiguous and fragmented resources. In our discussions we will use the following notation:

1) R - size of the constrained resource after static reductions

2) $\mathcal{O}$ - the system overlap factor

3) $\tau_i(t)$ - the sum of the CPU allocation factor for job i at time t

4) $\bar{\tau}(t)$ - the sum of the CPU allocation factors at time t

5) (a_i, p_i, r_i) - the description of a job i; where a_i is the active time, p_i is the passive time, and r_i is the amount of resource R which is requested.

Additionally we will make the following assumptions:

1) All jobs which are to be scheduled in an interval (t_0, t_1) are present at t_o. Jobs which arrive after t_o will not be scheduled until t_1.

2) No job may request the entire resource, i.e. for all i, $r_i < R$.

3) Dynamic reductions appear as schedulable jobs with the description $(0, p_d, r_d)$ where p_d is the length of time of the dynamic reduction and r_d is the size of the dynamic reduction.

4) The system overlap factor, $\mathcal{O}$, is a constant, $\mathcal{O} \leq 1$, for two or more jobs in execution, with equality holding when there is only 1 job in the system.

Note: If a job is processing when we choose to begin scheduling we will treat that job as a new job with new parameters. Although the resource requirement will remain unchanged, the active and passive times will be modified to reflect the fact that some processing has already occurred. If at time t_2 we investigate the system we will find that the amount of active time used by job i in the interval (t_1,t_2) is given by: $\int_{t_1}^{t_2} \tau_t \, dt$. Using this (or perhaps by simply keeping an internal record of the active time used by the job), we may calculate the new values of a_i and p_i, as

$$a_i' = a_i - \int_{t_1}^{t_2} \tau_t \, dt$$

$$p_i' = a_i'\left(\frac{p_i}{a_i}\right) \quad .$$

The description of job i then becomes (a_i',p_i',r_i) and we treat it as a new job for the purpose of scheduling the next interval. Having set the stage we now proceed to develop a best fit scheduling technique for a system with a single constrained resource.

We recall from chapter 2 that the minimum flowtime, F_{min}, for a set of schedulable jobs will be achieved, if $\overline{\tau}(t) = \mathcal{O}$ for all t in the execution interval. Intuitively, this makes sense since it simply says that the best that we can do is to use every available instant of processor time. Our ultimate goal is then to schedule a set of jobs in the interval (t_o,t_1) such that

$$\overline{\tau}(t) = \mathcal{O} \qquad t \in (t_o,t_1) \tag{1}$$

and

$$\sum_{i_t=1}^{k} r_{i_t} = R \ . \tag{2}$$

The desirability of this goal is obvious, since it would imply that the CPU was used to its fullest extent while no resource went idle. The attainability is at best elusive, and thus we will replace (1) and (2) with the minimization constraints

$$\min(\mathcal{O}-\overline{\tau}(t)) \qquad t \in (t_o,t_1) \tag{3}$$

$$\min(R - \sum_{i \in E_{s_t}} r_i) \quad t \in (t_o, t_1) \; . \tag{4}$$

3.1 Scheduling Contiguous Memory Resources

The scheduling problems encountered in dealing with contiguous memory resources are somewhat analogous to those presented by the cutting stock problem [5]. A contiguous memory resource, as defined in chapter 2, is a memory resource which must be assigned in a single contiguous portion. This is similar to the case of a customer who orders a piece of sheet metal from a foundry. Just as the sheet metal customer would be unable to use several small pieces in place of the one large piece that was originally ordered, the user of a contiguous memory resource is unable to use small pieces of non-contiguous memory to satisfy his request.

Let us assume for the moment that both the foundry operator and the computing center are faced with the same problem. Both must determine the method of sectioning a resource, sheet metal for one, CORE for the other, in a manner which minimizes the wasted resource. We view the resource requests as two-dimensional requests, that is length and width of sheet metal or amount of memory and period of time it is needed. The two organizations turn to linear programming techniques and use the cutting-stock solution obtained. The foundry operator applies his solution, minimizes his waste, and smiles all the way to the bank. The management of the computer center, on the other hand, has discovered a truly unsettling result. As they begin to divide their resource among the users, they find that the user must now have his portion of the resource for a longer period of time.

We have seen in chapter 2, the cause of this elongation. It is, in fact, a direct result of the portioning of the resource among more than one user, or multiprogramming. The results of the cutting stock problem could have been used if the user or the computer center management could have guessed the actual length of time that the memory resource was needed. Unfortunately, the actual length of time is dependent not only upon the user job, but, as we have seen, upon the nature and number of other jobs being concurrently processed. Having had this exit barred, we must now turn to another.

Codd [4] has presented a technique for multiprogram scheduling which suggests a solution to our problem. His method is sufficiently complex and the results are important enough to warrant a detailed discussion of Codd's method. From our point of view the primary limitation to Codd's method is that it is based on elapsed time. As we have seen, the elapsed time of a job is a function of the nature and number of other jobs in the system. Any _a priori_ estimate of the elapsed time of a job is more likely than not to be in error. Some [3] would argue even as to the appropriateness of active time, since active time is dependent on various idiosyncracies of the program such as amount of input or number of iterations required to find a root of a polynomial. We will answer by saying that even if the active time is only a least upper bound, the estimate of elapsed time which can be calculated at execution time will be much more accurate than an elapsed time estimate which may be too short when five programs are running and too long when two programs are running. We proceed with a discussion of Codd's multiprogram scheduling algorithm.

The scheduling algorithm is designed to deal with two types of facilities, space-shared facilities and time-shared facilities. We will deal only with space-shared facilities, since the aspects of the time-shared facility, the CPU, are included in our definitions of the CPU allocation factor, $\tau_i(t)$. Each job which enters the system appears as a pair $(e\ell_i, r_i)$ representing the elapsed time and resource request, respectively. The pair $(e\ell_i, r_i)$ will be treated as a rectangle and the goal is to place these rectangles within the larger rectangle whose dimensions are limited by the total amount of the resource, R, and the unbounded time axis. We will use the terms rectangle and job synonymously.

The rectangles are placed in the time-space area using five rules. These rules will be briefly described.

Rule 1. The resource is bounded by two load limits, B and b. B, the upper limit, is equal to R the total amount of resource, b, the lower limit is set arbitrarily. Each placement must satisfy the following three criteria.

1) Before program P is placed the sum of the resource requests must be less than b.
2) After P is placed, the sum of the resource requests must be less than or equal to B.
3) The rectangle which represents P must not intersect another rectangle.

Rule 2. Each program must be placed in the leftmost eligible position. Inserted idle time is not necessary nor desirable.

Rule 3. A program may not be placed in a position which would entail more than one schedule test.

Rule 4. Each program must be placed such that its upper or lower edge is completely bordered by another program or the upper or lower edge of the resource axis.

Rule 5. Once a program is assigned a portion of space, it may not be moved to another portion. In other words execution time relocation is not allowed.

Using these rules, Codd has proceeded to develop a scheduling algorithm. Two concepts are basic to this algorithm. They are the concept of a step and a pyramid. We will deal with the pyramid concept first. The concept of a pyramid is a direct result of the application of Rule 4. A pyramid consists of a set of rectangles which have been stacked upon one another. As a consequence of Rule 4, each layer must be less than or equal in length to the previous layer. The base of the pyramid must lie on either the upper or lower boundary of the resource map and must be greater than or equal to any other level of the pyramid. An exception to the rule that each layer be less than or equal to the preceding layer is made when an upper and lower pyramid meet. Several examples of pyramids are shown in Figure 3.1.

The concept of a step is provided as a method of dealing with the discontinuities which arise from the creation of a pyramid. A step is associated with either an upper or lower pyramid and consists of a start time and a length. Using a step, we may then determine whether a program with a given elapsed time can be used to further the construction of a pyramid without violating Rule 4. Since the addition of each program to the schedule will change the values and number of steps we present the algorithm which govern their creation. The algorithm applies equally to upper or lower levels but we will present it only from the lower level viewpoint. The upper level algorithm may be obtained by interchanging upper and lower in each step of the algorithm.

We will define the top of a pyramid as that level which is completely exposed, i.e. has no level above it. The length of the top level will be called d_t. The null pyramid is one in which the lower level of the resource axis is exposed. The length of the top level of a null pyramid is $d_t = \infty$. We will use s_p and d_p to represent the starting time and duration of job P. The

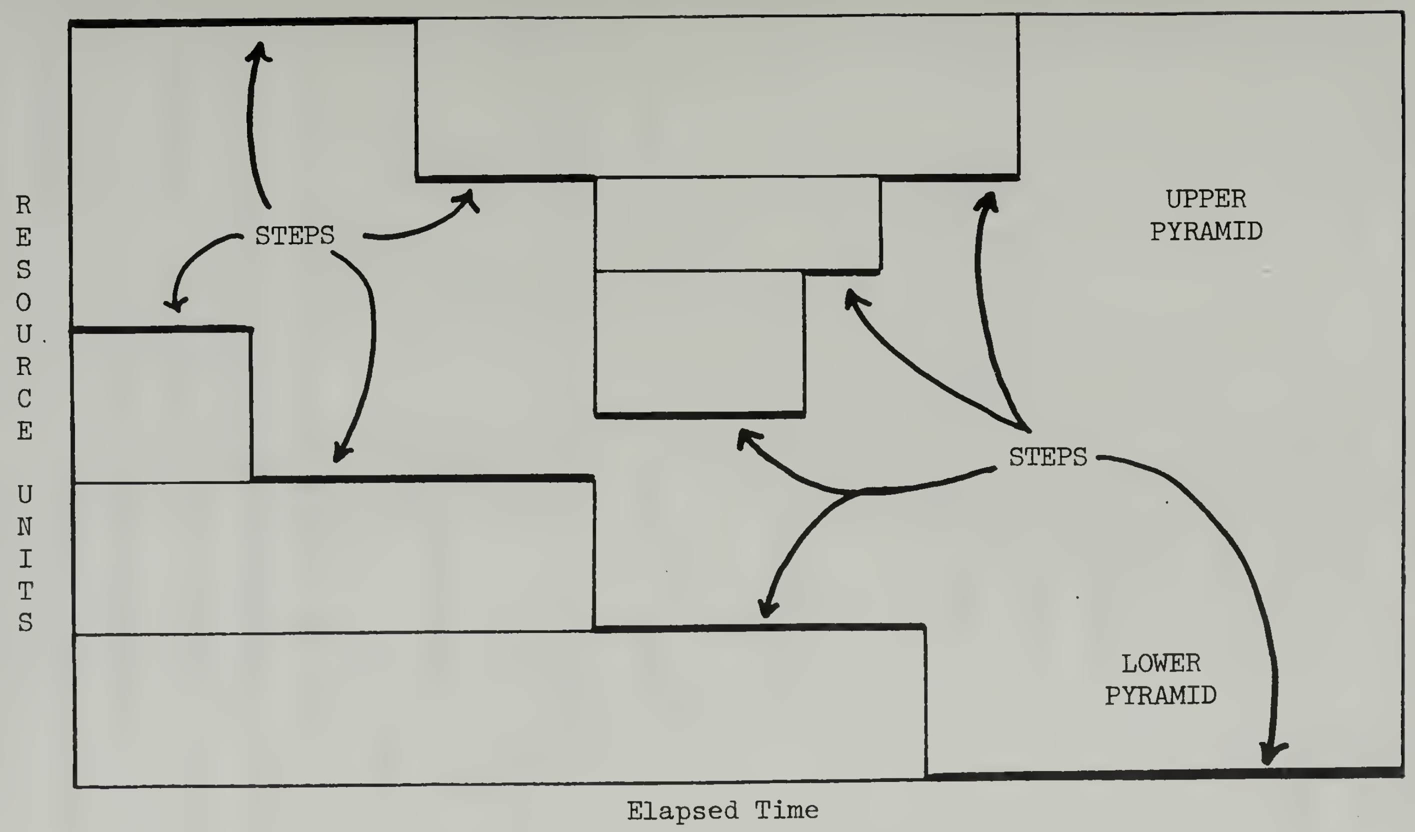

FIGURE 3.1 Some examples of pyramids

symbol x_t will represent the start of the level and the use of a prime will denote a level in the opposite list. Upon placement of job P, we will create at most three new steps from the step (x_t,d_t). They are

$$(x_t,s_p-x_t) \tag{5}$$

$$(s_p,d_p) \tag{6}$$

and

$$(s_p+d_p,x_t+d_t-s_p-d_p) \ . \tag{7}$$

If the step immediately preceding (x_t,d_t) in the opposite list meets the condition $x'_{t-1}+d'_{t-1} > x_t$, then we will replace the step (x'_{t-1},d'_{t-1}) with the two steps

$$(x'_{t-1},x_t-x'_{t-1}) \tag{8}$$

and

$$(x'_t,d_{t-1}-x_t+x'_{t-1}) \ . \tag{9}$$

Since jobs can only be scheduled at the termination of some other job, then s_p must equal some x already in the list. If, in fact, $s_p = x_t$ then equation (1) results in a step of zero duration and may be dropped. In general, if the duration of any step becomes zero, that step may be removed from the list.

Using the step list which has been described above and a list, initially empty, of jobs which have been scheduled, Codd now proceeds to schedule the next unscheduled job. He locates the first step which is long enough to contain the job. If the current resource level usage is such that the inclusion of the job would exceed the upper limit, B, then the next starting time on this step is found. The search on this step will continue as long as there remains sufficient time to complete the job. If no such position is found, then the next step is examined. When a job is finally placed in the schedule, the step list is updated as described above.

We have already discussed the problems inherent in the use of elapsed time for scheduling purposes. It is important to note that even the use of programmer-supplied active and passive times with the CPU allocation factor may not provide an accurate estimate of elapsed time. Through the enforcement of termination upon exceeding the specified active time, it can usually be guaranteed that the calculated elapsed time will not be exceeded. Unfortunately, there is no guarantee that the program will not use less than the requested amount of active time.

Having presented Codd's algorithm, we will now discuss a few modifications in an attempt to improve the algorithm. Our primary modification will be the use of computed run times based on the programmer or system supplied active and passive times. From our discussion of multi-programming effects we have noted that a prolongation in run time occurs whenever a job receives less than the maximum CPU allocation factor which it can use. The maximum CPU allocation factor, $\tau_{i_{max}}$, is defined as ratio of active time to execution time or

$$\tau_{i_{max}} = \frac{a_i}{a_i+p_i} \quad . \tag{10}$$

Thus when the number of jobs being executed, n, is such that $\frac{1}{n} \geq \tau_{i_{max}}$, then job i will not suffer a prolongation of run time. Since we wish to make extensive use of the previous work we must take this prolongation effect into account.

We use these results by first ordering our list of unscheduled jobs in decreasing execution time, a_i+p_i, within decreasing $\tau_{i_{max}}$. The reason for this ordering is two-fold. First, we wish to keep the longer running jobs to the outside of the resource area. By maintaining the list of unscheduled jobs in decreasing order of execution time, which, by definition, is the minimum time for the job to complete, we guarantee that even if another job is placed above the first job and both are processed at $\tau_{i_{max}}$, the first job will not complete before the second job. Second, since jobs with equal execution times are maintained in order of decreasing $\tau_{i_{max}}$, we insure that even if they are processed at less than $\tau_{i_{max}}$, the first job will suffer greater prolongation of run time than the second job.

To illustrate, let two jobs k and ℓ with equal execution times appear in the order k, ℓ in the list of unscheduled jobs. The execution time, E, is then given by

$$E = a_k/\tau_{k_{max}} = a_\ell/\tau_{\ell_{max}} \ . \tag{11}$$

Since $\tau_{k_{max}} \geq T_{\ell_{max}}$, then $a_k \geq a_\ell$. The method used to determine $\tau_i(t)$ insures that $\tau_k(t) = \tau_\ell(t)$ or $\tau_k(t) > \tau_\ell(t) = \tau_{\ell_{max}}$. If $\tau_k(t) = \tau_\ell(t)$, then job ℓ will finish first, since $a_k > a_\ell$. If $\tau_k(t) > \tau_\ell(t) = \tau_{\ell_{max}}$, then job ℓ finishes first or at the same time, since $a_\ell/\tau_{\ell_{max}} = E$ is the minimum execution time and $\tau_k(t) \leq \tau_{k_{max}}$ which would require job k to finish either at the same time or later than job ℓ.

Our strategy is then to move down the unscheduled list inspecting each job. Using the resource usage parameter, r_i, of the 3-tuple which defines the next job, we determine whether the inclusion of this job in the current schedule will exceed the amount of available resource, R. If this job will fit, resource-wise, then we must check to see if it can be used to build the next layer of a pyramid. Since we are dealing with a dynamic system, we will consider the scheduling interval to be that interval between the present moment (set to zero at the beginning of each interval) and the earliest termination of a job executing in that interval. We have already discussed the treatment of jobs which extend into the next scheduling interval.

At the beginning of each scheduling interval a step list will exist which contains a list of the eligible positions for a job and the length of that position. The elements of the step list are of two types: 1) those steps which are formed from the exposed sections of pyramids whose bases lie on the lower level of the resource map, termed lower level steps; 2) those steps which are formed from the exposed sections of pyramids whose bases lie on the upper limit of the resource map, termed upper level steps. This step list will contain either two elements of the type (0,t,X) where t is the duration of the step and X represents a lower or upper step, or it will contain no elements of the type (0,t,X). From the construction of the pyramids, only those jobs which form the highest (lowest) levels of a lower (upper) pyramid may terminate in a given scheduling interval. As a result, the resource which is freed is contiguous and can be represented by the two step list elements $(0,t_U,U)$ and $(0,t_L,L)$. Because of our definition of the scheduling interval, the pyramids which exist at the beginning of any interval are always left justified.

Initially, the step list contains two entries: $(0,\infty,L)$ and $(0,\infty,U)$. The first job is then guaranteed to fit on an available step. When this job is placed in the schedule, then the step list is updated by replacing the entry $(0,\infty,L)$ with the entries $(0,a_1/\tau_{1_{\max}},L)$ and $(a_1/\tau_{1_{\max}},\infty,L)$. The resource usage for this job is added to the current resource usage level, R_c. The next job which satisfies $r_i \leq R-R_c$ is then located. Upon placing this job in the schedule, we now recalculate the $\tau_i(t)$ for the two jobs and replace the step list entry $(0,a_1/\tau_{1_{\max}},L)$ with the two entries $(0,a_2/\tau_2(t),L)$ and

$\left(\frac{a_2}{\tau_2(t)}, \frac{a_1 - \frac{a_2}{\tau_2(t)}(\tau_1(t))}{\tau_{1\max}}, L\right)$. The step list entry for $\left(\frac{a_1}{\tau_{1\max}}, \infty, L\right)$

is replaced by $\left(\frac{a_2}{\tau_2(t)} + \left(a_1 - \frac{a_2}{\tau_2(t)}\right)\left(\frac{\tau_1(t)}{\tau_{1\max}}\right), \infty, L\right)$.

In general, the method for updating the step list is identical to that proposed by Codd. Specific differences, which are mainly reductions in the amount of work done, are a result of the restriction that we may only schedule jobs on the two steps which start at 0, namely $(0,t_U,U)$ and $(0,t_L,L)$. Since the size of the steps are determined by the jobs which are being run during the scheduling interval, we need not maintain the steps which do not fall in the current scheduling interval. Thus, we need only know the length of the last scheduling interval and the CPU allocation factor, $\tau_i(t)$, for each job during that interval. From this information we may then determine the amount of active time remaining for those jobs which are present at the beginning of this new interval.

To illustrate the above techniques, we will use them to schedule a group of jobs. The jobs to be scheduled are shown in Table 3.1. Note that we have provided $\tau_{i\max}$, the maximum CPU allocation factor, instead of the passive time, p_i. We will schedule these jobs on a contiguous memory resource which is four resource units large. The overlap factor is assumed to be equal to unity. We further assume that no jobs enter the system after the first scheduling interval.

We start by taking the jobs P_1,P_2,P_3, and P_{10}. The resource usage level is now four units. Upon calculation of the $\tau_i(t)$ for each of the jobs, and dividing the active time for each job, we find the job P_{10} will finish in 37.08 seconds. When job P_{10} completes, we check the unscheduled job list and find that no job in the list will fit in the remaining resource. We recalculate the $\tau_i(t)$ and find that this new scheduling interval will end at 212.79 seconds. At the beginning of the third interval, we find that the resource usage level is only two units. Examining the unscheduled job list we find that P_4 will fit in the remaining resource. In the fourth interval we find that the next job, P_5, will not fit on the lower step which is above P_2. To accommodate P_5 we schedule it on the upper step. The process

Job	Active time (secs)	$\tau_{i\max}$	Resource Request	Execution Time (secs)
1	60	.2	1	300
2	100	.66	1	150
3	80	.6	1	133
4	40	.3	2	133
5	50	.4	2	125
6	40	.4	2	100
7	30	.3	3	100
8	30	.5	3	60
9	20	.9	2	22
10	10	1	1	10

Table 3.1. Scheduling Data

in the above manner until all of the jobs have been scheduled. The final schedule is graphically displayed in Figure 3.2. A Gantt chart has been provided in Figure 3.3 showing the CPU allocation factors used in each scheduling interval.

We have thus demonstrated that there exist methods for deriving best-fit type schedules for the class of constrained contiguous memory resources.

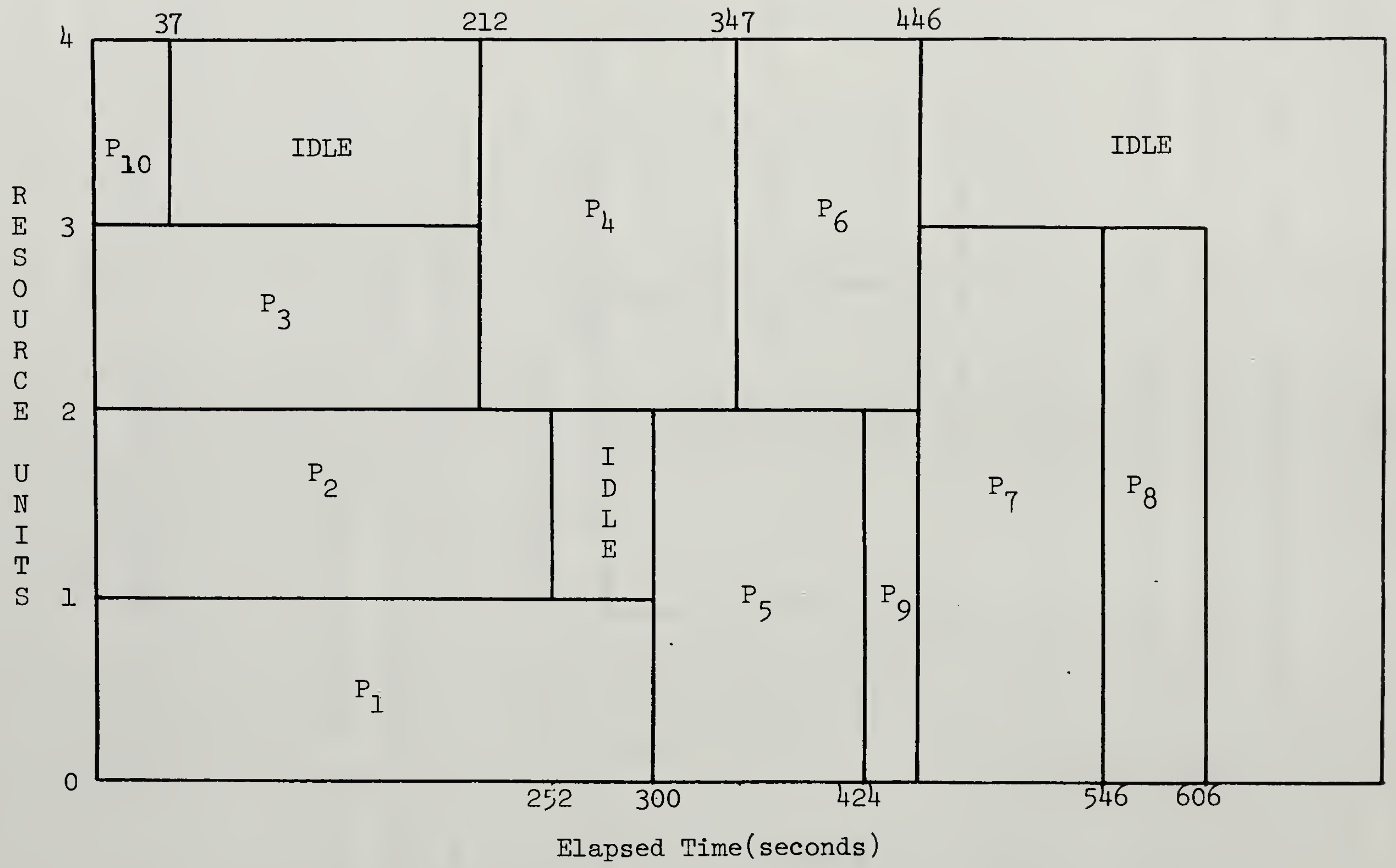

FIGURE 3.2 Contiguous memory schedule for jobs in Table 3.1

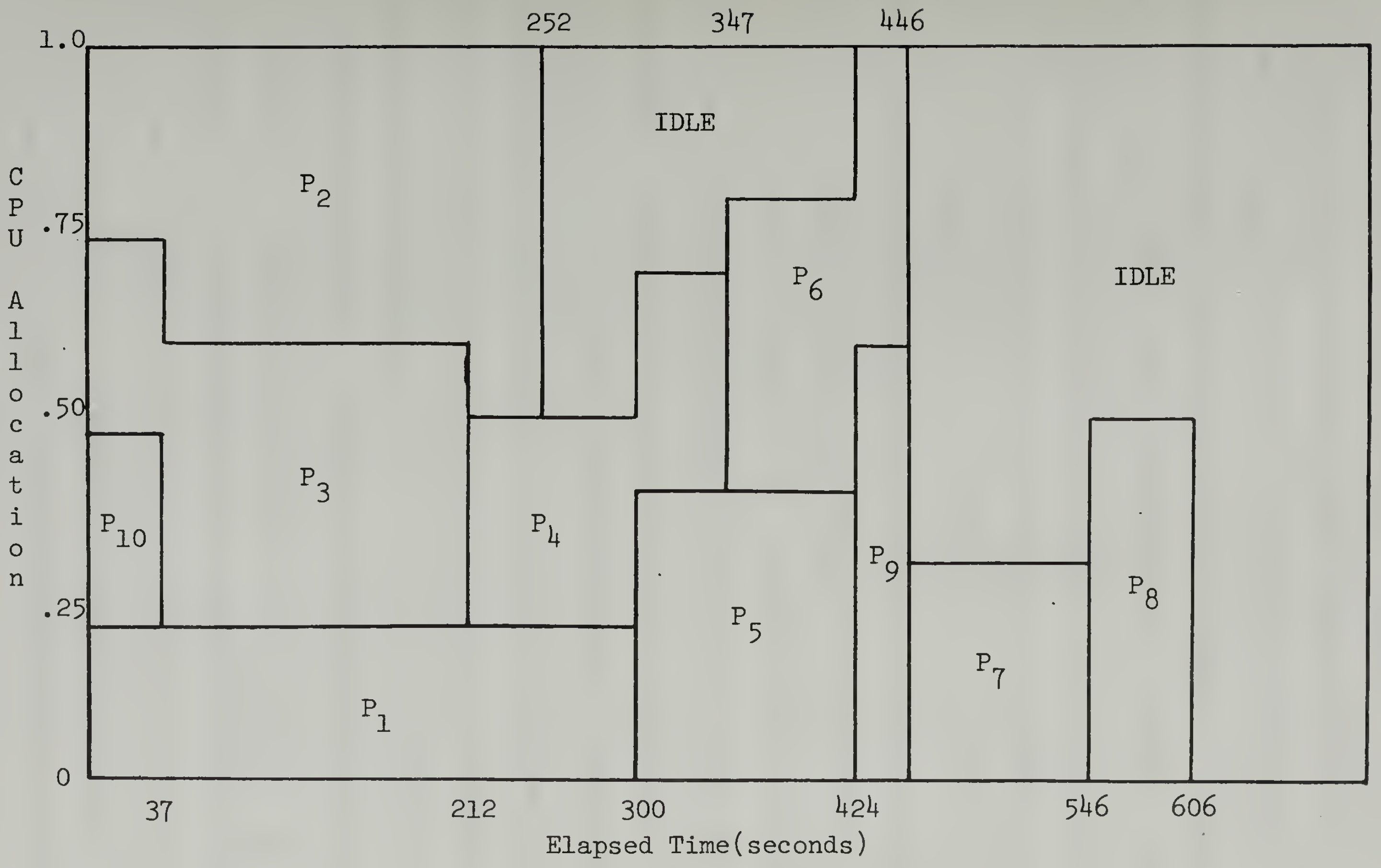

FIGURE 3.3 Gantt chart for schedule shown in Figure 3.2

3.2 Scheduling Fragmented Memory Resources

We have seen in the previous section that best-fit schedules do exist for systems in which there is a single constrained memory resource of the contiguous type. A logical question is then whether these methods may be applied to the problem of determining a best-fit schedule for fragmented memory resources. Examples of a fragmented memory resource are magnetic tape drives or the CORE in a system which operates under a paging philosophy. The results can, in fact, be applied to the problem of fragmented memory resources. But, unfortunately, to accomplish this we would have to remove the freedom of allocation which is inherent in the definition of fragmented memory resources and treat them as contiguous resources. This seems hardly worth the effort since the freedom of allocation in fragmented memory resources is an advantage which we do not care to cast aside.

We must not be hasty, however, in discarding the techniques which have been developed for contiguous memory resources. Our principal problem is still one of covering the large resource-time rectangle, which represents the computer system, as completely as possible with the smaller resource request-time rectangles which represent the jobs to be run. We find that by employing the definition of fragmented memory resources, a simplification exists which will permit us to make use of previous results. In dealing with contiguous memory resources we were required to manipulate rectangles whose resource dimension was always equal to the resource usage parameter, r_i, of the job. Fragmented resources, on the other hand, permit us to use up to r_i rectangles to represent the resource requirements of a job.

Once the freedom to use multiple rectangles has been achieved, we find that the pyramid restriction is no longer necessary. We will show the reasoning behind the removal of this restriction and the resulting simplification in the method.

Let us assume that three jobs described by the 3-tuples (a_j, p_j, r_j), (a_k, p_k, r_k) and (a_ℓ, p_ℓ, r_ℓ) have been scheduled for concurrent processing. Furthermore let us assume that the jobs have been assigned adjacent resource units. That is, units 1 through r_j have been assigned to job j, units r_{j+1} through r_{j+k+1} to job k and units r_{j+k+1} through $r_{j+k+\ell}$ to job ℓ. The remaining $R-r_j-r_k-r_\ell$ resource units are unassigned. Now if, in violation of the pyramid rule, job k terminates before job ℓ, we will have two groups of free resource units, one with r_k free units and one with $R-r_j-r_k-r_\ell$ free units. For contiguous resources we would now be limited by scheduling jobs with resource requirements, $r_i \leq \max(r_k, R-r_j-r_k-r_\ell)$. However, since we are dealing with a fragmented resource we may schedule a job with a resource request of $r_i \leq R-r_j-r_\ell$, the entire amount of free resource.

Suppose that the job we wish to schedule has a resource usage of $r_i \leq R-r_j-r_\ell$. If $r_i \leq \min(r_k, R-r_j-r_k-r_\ell)$, then we may schedule it in either area. If $r_i \geq \max(r_k, R-r_j-r_k-r_\ell)$, then we will use two areas, one which is $r_i-\max(r_k, R-r_j-r_k-r_\ell)$, and the other which is $\max(r_k, R-r_j-r_k-r_\ell)$. We may use any number of groups to arrive at the required number of resource units. We see then that we are not concerned with the location of the free units, but with the total quantity of the free resource units. As a consequence, we may remove the pyramid restriction when dealing with fragmented resources.

Having removed the pyramid restriction, we are now able to treat the scheduling of a job. Our algorithm for scheduling fragmented resources will be as follows:

(1) Determine the length of the current scheduling interval by examining all jobs which are currently processing and determining which one will finish first and when it will finish.

(2) Select a candidate job which has a resource request commensurate with the free resource units.

(3) Calculate the tentative CPU allocation factors for this job and the others which are currently being processed and using the tentative $\tau_i(t)$, determine the length of the next scheduling interval and the amount of free resource at the end of the interval.

(4) Note the tentative amount of free resource at the end of the new interval if it is greater than or equal to the current amount.

(5) After checking every job which will fit the resource availability, select the one which will result in the greatest amount of free resource at the end of the next scheduling interval.

In order to demonstrate the use of the fragmented memory scheduling algorithm, we present the following example. The jobs which are to be scheduled, their active time, $\tau_{i_{\max}}$, and their resource requirements are shown in Table 3.2. We will assume that all of the jobs are present at the beginning of the first interval. The jobs have been ordered by decreasing resource request and decreasing execution time. As in the example presented for the contiguous memory case, we assume that $\Theta = 1$ and that no jobs enter the system after the beginning of the first scheduling interval. The fragmented memory resource has an available resource amount, $R_i = 7$.

Since at the start of the first interval, the resource usage is zero, we may take any job. We will choose P_1, since its completion will release the most resource. There are now 2 resource units left. Upon examining the jobs which require 2 or less resource units we find that P_6 will finish at the same time as P_1 and 7 units will be available. At 133 seconds, when P_1 and P_6 terminate, we select P_2 with a request of 4 units. To fill the remaining 3 units

Table 3.2. Scheduling data for fragmented memory resource example

Job	Active time (secs)	$\tau_{i_{max}}$	Resource request	Execution time (secs)
1	80	.6	5	133
2	10	1	4	10
3	100	.66	3	150
4	50	.4	3	125
5	60	.2	2	300
6	40	.3	2	133
7	30	.5	2	60
8	40	.4	1	100
9	30	.3	1	100
10	20	.9	1	22

we will select P_3. Upon termination of P_2 at 153 seconds, there are four free resource units. We select P_4 for execution, since its termination will release 3 units. P_8 is then assigned to the remaining 1 unit of resource. Upon determination of the $\tau_i(t)$ we find that P_8 will terminate at 275 seconds. At that time we will schedule P_9, which has a resource request of 1 unit. We proceed in this manner until all of the jobs have been scheduled. The entire schedule is presented in Figure 3.4. The $\tau_i(t)$ used in each schedule are shown in the Gantt chart of Figure 3.5.

We have produced two algorithms for use in systems with a single constrained resource. The first is designed to deal with contiguous memory resources, and the second is directed towards the problem of a fragmented memory resource.

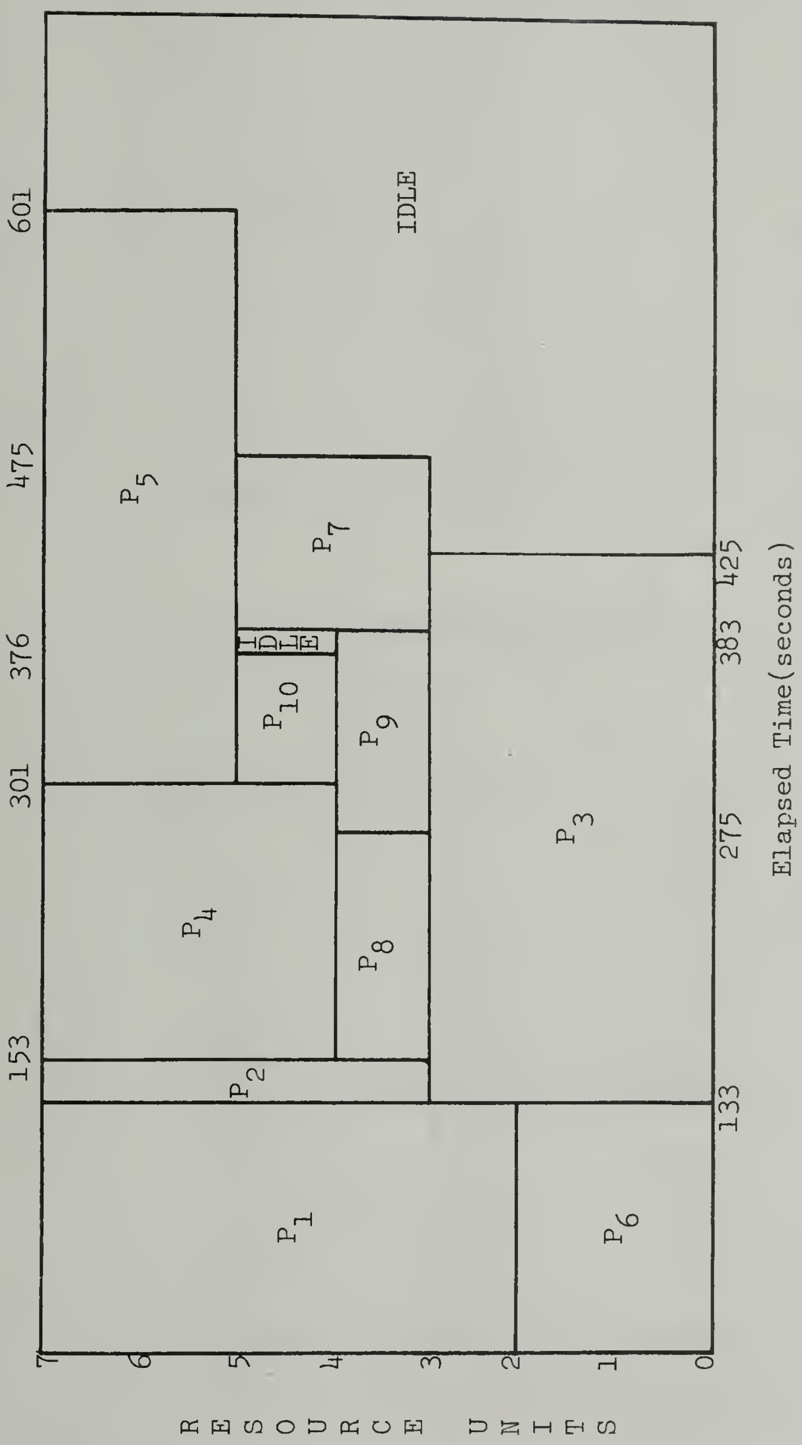

FIGURE 3.4 Fragmented memory schedule for jobs in Table 3.2

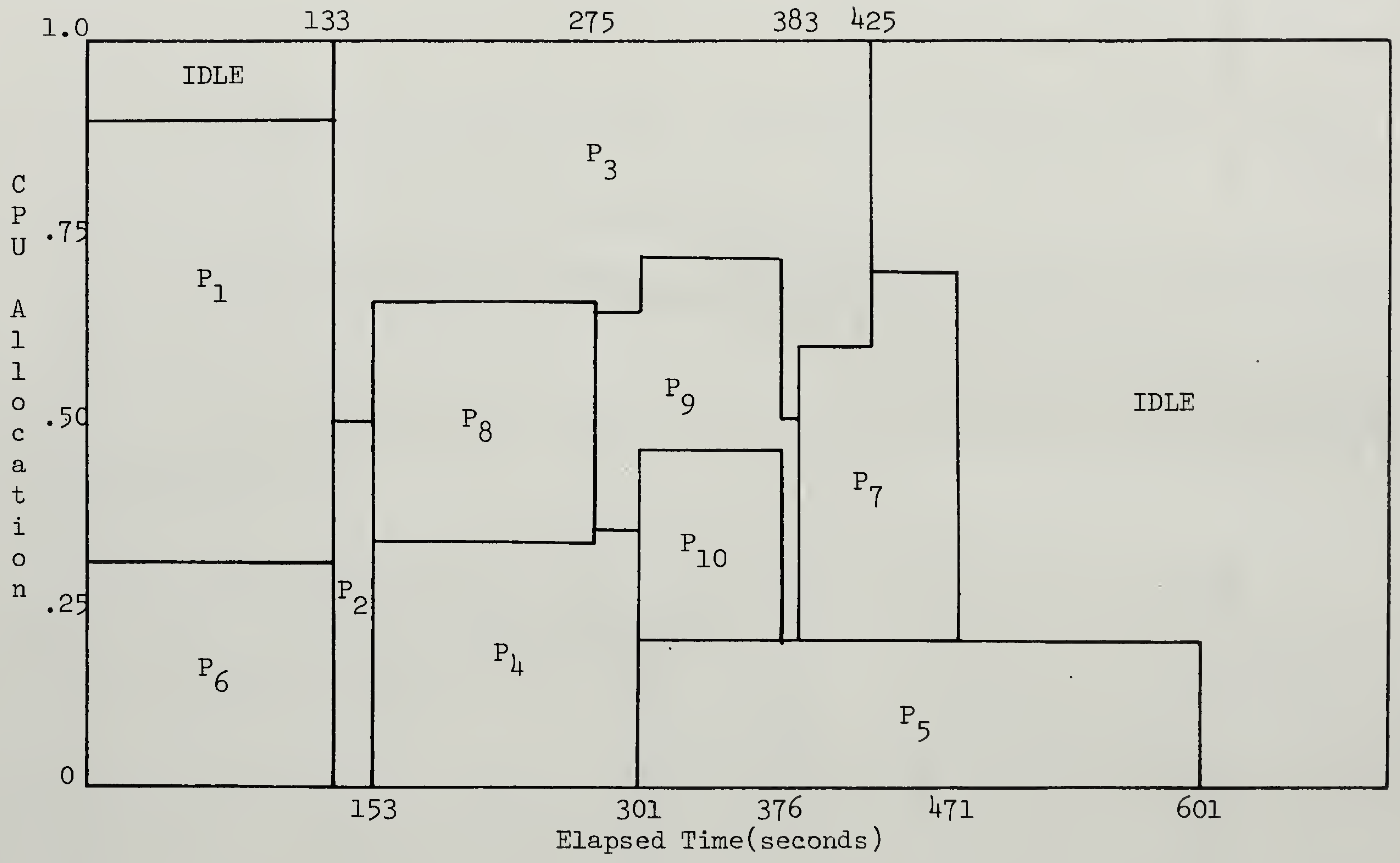

FIGURE 3.5 Gantt chart for schedule shown in Figure 3.4

4. MULTIPLE MEMORY CONSTRAINTS

We have seen that when there is only a single constrained memory resource, certain techniques may be used to derive schedules which will tend to minimize the amount of idle resources. In some systems this single resource approach may be all that is necessary. Computer centers, such as those sometimes found in academic environs (which are subject to workloads involving infrequent requests for magnetic tape drives) will find that the critical resource is CORE. In centers of this type the scheduling is most sensitive to resource requests for CORE and, as a result, the method for scheduling a single contiguous memory resource would be more than adequate. Other centers may find that a fragmented memory resource is the dominating factor, and will use the fragmented memory resource technique.

The important point in the above cases is that only the dominating constrained memory resource need be considered in effectively scheduling jobs. We direct our attention in this chapter to those systems in which there are more than one constrained memory resource.

In systems with multiple constrained memory resources we have two objectives. First, we hope to schedule the jobs in a manner which will minimize the idle portion of each memory resource. Second, we desire a schedule which will tend to balance the queue lengths between the various resources (that is we want to avoid the formation of large queues that naturally result when one resource is dominant in scheduling). The problem of achieving these two goals and thus providing a well balanced schedule is a difficult one. We have not yet found a general solution to this problem and in the following we will only describe the various facets of the general solution presenting possible points of departure for future research.

Systems which have multiple constrained memory resources are dynamic. Owing to variations in the workload, the critical constrained resource (the one which most affects overall performance) may change. These variations can cause a shift in emphasis from a fragmented resource to a contiguous resource, or even result in the system becoming a single constrained memory resource system. Our method must be sensitive to these variations if it is to be effective.

In viewing a system with multiple resource constraints, we consider the system to be an n+1 dimensional solid. This solid is composed of the real time axis and n other dimension which are the n different resources of the system. Jobs to be scheduled on this system may then be viewed as small n+1 dimensional solid which are to be packed in the larger solid which is the system.

Denning [3] has suggested that "resource allocation is then a matter of balancing memory and processor demands against equipment." While his orientation is one of a paged system, and is concerned with determining the best set of pages to maintain in memory, the balancing concept appears to be a useful one. The balancing act which we must perform is, indeed, a tricky one. In selecting jobs which would return the system to a "balanced" state, we must insure that their selection will not unbalance the system from another direction.

Our most obvious approach at this point is to use the same methods which were developed for systems with single constrained memory resources. We define a set of tolerance functions, $T_i(n,D)$. The tolerance functions are functions of n, the number of jobs in the queue which are requesting the resource i, and D, the total demand of those jobs for resource i. Each tolerance function is designed to produce a value which indicates how much idle time we are willing to accept on a given resource. For a resource which is not constrained, then $T_i(n,D) < 0$ for all values of n and D. Critical resources, on the other hand, will have tolerance functions which are always large.

We are now ready to attempt to schedule jobs. The candidate job is tentatively placed in the schedule and the amount of idle resources caused by the placement of this job are measured against the tolerance functions. If any of the tolerance functions are exceeded, we note the amount, the job is deferred, and the search continues until we find a job which meets the tolerance function restrictions. If no job can be found which meets the restrictions we then choose the one which least exceeds the tolerance functions.

The concept of a tolerance function provides us with a rather convenient solution to our scheduling problem. Unfortunately, the _a priori_ determination of the exact form of the tolerance function for a given resource presents a somewhat formidable task. This problem results from the fact that the tolerance function may depend on the nature of the resource, its value to the system, and the importance of the users which are demanding this resource, in addition to the size of the queue and total demand for this resource. However, because of the method in which the tolerance functions are used, the tolerance functions can be easily changed and thus can be tuned to reflect local needs and desires.

Suppose that a particular tolerance function is suspected to be a linear function of the form $T_i(n,D) = An+BD+C$, where A, B, and C are unknown coefficients and n and D are determined from system values. Initially, we might assign values of $A = 1$, $B = 1$, and $C = 0$. This assignment has the effect of treating resource demand and queue size as equally important terms and since the value of this function is always non-negative, we see that the resource will be treated as a critical resource. If the nature of the resource is such that it only becomes critical at some higher values of queue size and total demand, then we could assign some negative value to the constant, C. With a tolerance function of the form $T_i(n,D) = n+D-C$, we find that the resource is scheduled as a non-critical resource until the sum of the queue size and the resource demand exceeds the value of C, and then as a critical resource at higher values.

Similar adjustments can be made to the coefficients A and B. If we find that the nature of the workload is such that each job requests only a few units of the resource, then we might wish to make $T_i(n,D)$ more responsive to queue size by increasing the value of A. On the other hand, if we wish to be more responsive to jobs which demand large portions of the resource we might increase the value of B and thus cause the scheduling emphasis on total resource demand rather than on queue size. The above argument has been presented for a linear function. Similar methods can be used in cases where the tolerance function is non-linear.

We see then that, in lieu of an obvious relationship, a rough function which meets some of the operating system's requirements can be provided. Then through subsequent analysis additional terms may be added and changes made to the coefficients to improve the agreement of the tolerance function with the needs and desires of the computer system, and provide effective scheduling of the constrained memory resources.

We present the following example to demonstrate the method of scheduling a system in which there are two constrained memory resources. For this example, we will use one resource, C, which is contiguous and one resource, F, which is fragmented. The amounts of available resource will be $R_C = 4$ and $R_F = 7$. The overlap factor used will be unity. We will assume that the tolerance functions have been previously determined to be:

$$T_C(n,D) = n+D$$

$$T_F(n,D) = n+D \ .$$

The jobs which we will schedule are shown in Table 4.1. They are the same jobs we have used in prior examples with the exception that this time we will be dealing with two resource requests instead of one. The jobs are ordered as they were for the contiguous memory resource example.

Figures 4.1 and 4.2 depict the schedule in terms of the two memory resources. Table 4.2 shows the progress of the schedule. We will not discuss the creation of the schedule to any depth, since almost all of the calculations have been presented at one time or another. One item we will discuss is the decision made at 271 seconds into the schedule. At this point we had four jobs left to schedule. Upon inspecting the tolerance functions, we find that the fragmented memory is more critical ($T_F > T_C$). We chose to schedule P_3 instead of P_7, P_5 or P_8 since only this choice would reduce the critical fragmented resource usage more than it would the contiguous resource usage.

Table 4.1. Set of jobs for 2-resource example

Job	Active time (secs)	τ_i^{max}	r_C	r_F	Execution time (secs)
1	60	.2	1	2	300
2	100	.66	1	3	150
3	80	.6	1	5	133
4	40	.3	2	2	133
5	50	.4	2	3	125
6	40	.4	2	1	100
7	30	.3	3	1	100
8	30	.3	3	2	60
9	20	.9	2	1	22
10	10	1	1	4	10

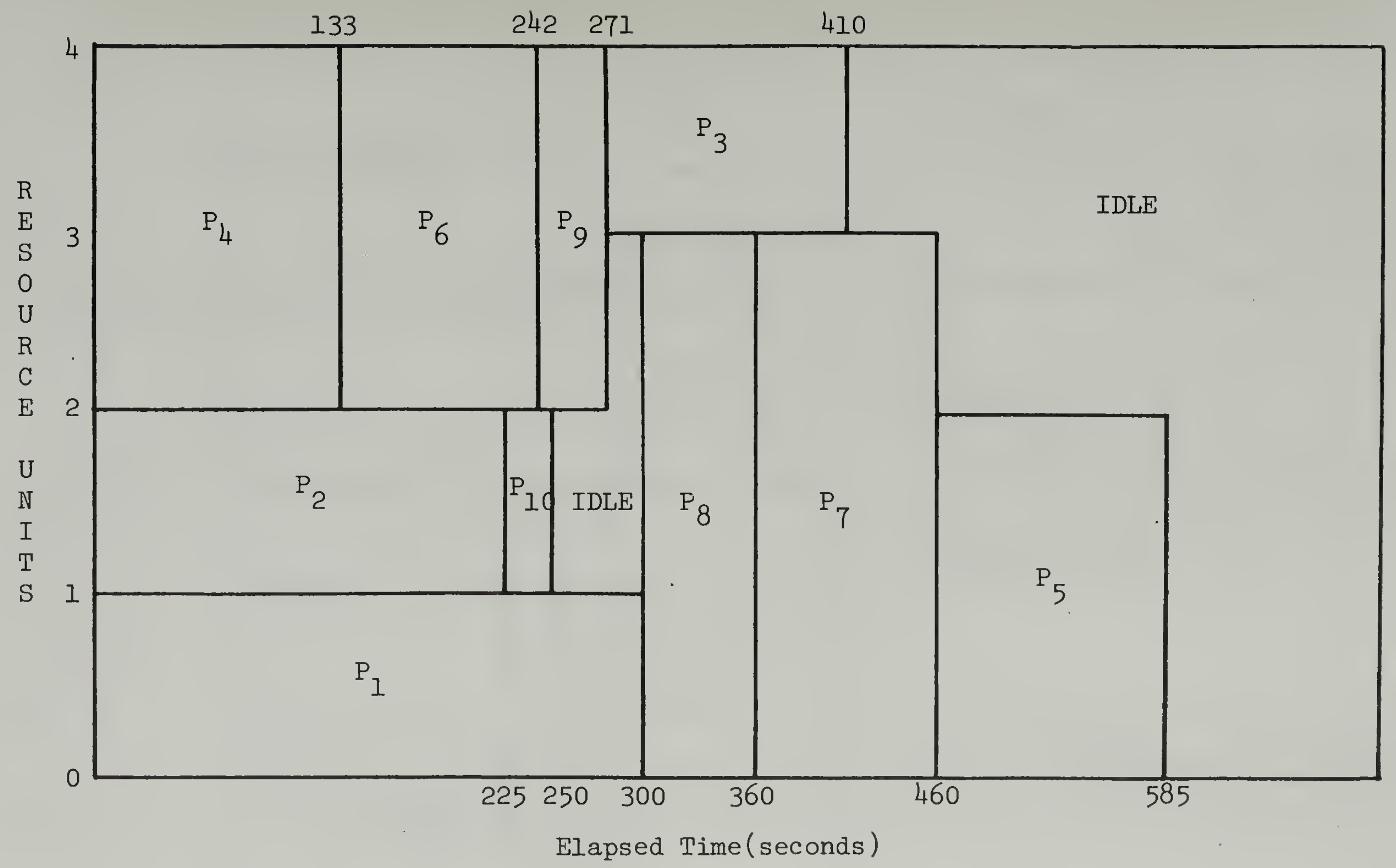

FIGURE 4.1 Contiguous resource usage for the schedule shown in Table 4.2

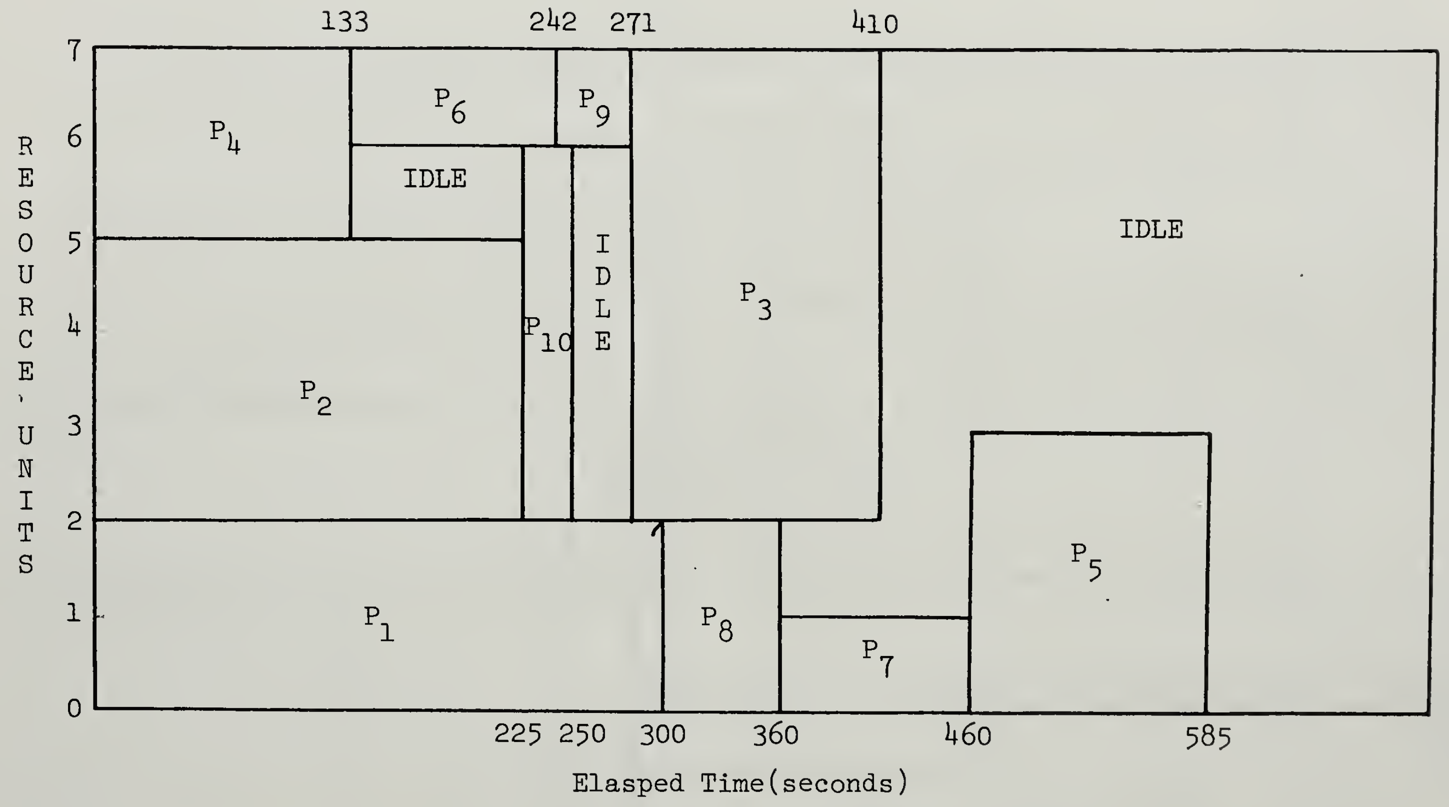

FIGURE 4.2 Fragmented resource usage for the schedule shown in Table 4.2

Table 4.2. Scheduling intervals for the 2 resource example. * denotes the first appearance of a job in the schedule.

Time	Job	Remaining Active Time	τ_i^{max}	$\tau_i(t)$	r_C	r_F	T_C	T_F
0	1*	60	.2	.2	1	2	26	31
	2*	100	.66	.5	1	3	24	27
	4*	40	.3	.3	2	2	21	24
133	1	33	.2	.2	1	2		
	2	33	.66	.4	1	3		
	6*	40	.4	.4	2	1	18	22
225	1	15	.2	.2	1	2		
	6	7	.4	.4	2	1		
	10*	10	1	.4	1	4	16	17
242	1	12	.2	.2	1	2		
	10	3	1	.4	1	4		
	9*	20	.9	.4	2	1	13	15
250	1	10	.2	.2	1	2		
	9	17	.9	.8	2	1		
271	1	6	.2	.2	1	2		
	3*	80	.6	.6	1	5	11	9
300	3	61	.6	.5	1	5		
	8*	30	.5	.5	3	2	7	6
360	3	31	.6	.6	1	5		
	7*	30	.3	.3	3	1	3	4
410	7	15	.3	.3	3	1		
460	5*	50	.4	.4	2	3	0	0

5. CONCLUSION

In the foregoing we have presented a system for examining the effect of interaction of multiprogrammed jobs. Using this system we then investigated a class of best fit scheduling techniques. We have seen that a best fit scheduling algorithm can be used to minimize the amount of idle resources and at the same time provide a nearly optimal flowtime. The reduction in idle resources which can be achieved at a given computing center will vary with processor speed, amount of resources and other factors which are intrinsic to a given computing center.

The ultimate result of using a best fit scheduling technique is to overlap CPU and memory usage and effectively compress the total flowtime for a set of jobs. This results in an expansion of the usage of a given resource, which promises to provide more efficient and effective resource utilization.

The class of best fit scheduling techniques is not intended to be the ultimate in scheduling techniques for multiprogramming systems. The most significant benefits of a best fit scheduling technique will be derived in those systems where the workload consists of numerous independent jobs since there is a potential to decrease the idle memory resources and increase through put as a result of collecting those small fragments of idle memory into a usable area.

The best fit techniques are not designed to operate in systems with large networks of dependent jobs having tightly defined deadlines. In these systems other techniques such as Cost Performance Monitoring-Program Evaluation Review Techniques (CPM-PERT) must be investigated [6]. Unfortunately, the CPM-PERT techniques can quickly become untractable when the networks grow large or become networks of networks. Future research will be required not only to improve the usability of CPM-PERT but also to provide insights into methods of combining CPM-PERT with the best fit scheduling techniques presented here.

This effort has resulted in one more step on the long journey toward total computer system optimization.

LIST OF REFERENCES

[1] Conway, R. W., Maxwell, W. L., and Miller, L. W., Theory of Scheduling , Addison-Wesley, 1967.

[2] Baskett, F., Browne, J. C., and Lan, J., "The Interaction of Multiprogramming Job Scheduling and CPU Scheduling", Proc. FJCC, AFIPS Press, Montvale, New Jersey, 1972, pp. 13-21.

[3] Denning, Peter J., "The Working Set Model for Program Behavior", Communications of the ACM, Volume 11, May, 1968, pp. 323-333.

[4] Codd, E. F., "Multiprogram Scheduling", Communications of the ACM, Volume 3 (1960), pp. 347-350 (Part 1 & 2) and pp. 413-418 (Part 3 and 4).

[5] Wagner, H. M., Principles of Management Science with Applications to Executive Decision , Prentice-Hall, 1970.

[6] Archibald, R. D. and Villoria, R. L., Network-based Management Systems, Wiley and Sons, 1967.

[7] Coffman, E. G. and Muntz, R. R., "Models of Pure Time-Sharing Disciplines for Resource Allocation", Proc. ACM, 1969, pp. 217-228.

[8] Belady, L. A. and Kuehner, C. J., "Dynamic Space-Sharing in Computer Systems", Communications of the ACM, Volume 12, May 1969, pp. 282-288.

[9] Bernstein, A. J. and Sharp, J. C. "A Policy Driven Schedule for a Time-Sharing System", Communications of the ACM, Volume 14, February 1971, pp. 74-78.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-678	2.	3. Recipient's Accession No.

4. Title and Subtitle

Cost Minimization in Computer Systems Subject to Multiple Memory Constraints

5. Report Date
October 1974

6.

7. Author(s)
Michael Austin Jamerson

8. Performing Organization Rept. No. UIUCDCS-R-74-678

9. Performing Organization Name and Address
University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address
Western Electric Company
28W615 Ferry Road
Warrenville, Illinois
and

13. Type of Report & Period Covered
Master of Science Thesis

14.

15. Supplementary Notes
University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, Illinois 61801

16. Abstracts

The reconciliation of service requests with memory resource availabilities presents a formidable problem for many computer service centers. When the sum of the service requests for a given resource exceeds the amount of the available resource, then that resource is constrained and scheduling techniques must be used to maximize the utilization of that resource. Two types of memory resource are considered, fragmented resources which can be allocated in pieces and contiguous resources which may only be allocated in one piece. An algorithm is developed which attempts to minimize the maximum flowtime for a set of multiprogrammed jobs while minimizing the idle resources.

17. Key Words and Document Analysis. 17a. Descriptors

Constrained Memory Systems
Multiprogram Scheduling
Minimization of Idle Resources
Scheduling Constrained Resources

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement: Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 52
	20. Security Class (This Page) UNCLASSIFIED	22. Price -----